Jun 25, 2019

A SIMPLE ENTITY COMPONENT SYSTEM (ECS) [C++]

Ever since first hearing about Entity Component Systems and their implications in game development, I've wanted to build one for my own usage and knowledge. There are a few examples that people have built and posted online (**1**, **2**), and there are a few full-fledged ones that can be used to build real games (**3** , **4**).

I liked different aspects of each of them that I studied, but I wanted to build my own and put my own spin on the problem, incorporating different elements from different examples. This is the result.

It is, admittedly and by design, simple. It's intended as a jumping off point for the curious.

UPDATE [2021-07-12]

In the two years since I wrote this, I've received many comments/emails about it, and it gets a lot of daily hits. I fear that people read it and take it as gospel when I never intended to be seen as an authority on the subject; I just wanted to write about something that I found interesting.

Since then my feelings on a few things have changed so I'd like to talk about them here briefly.

1) I don't like "Modern" C++ or really C++ in general.

I use some C++ features here that I would never use today as I've gone back to trying to write simple code. I will either use C++ with a limited feature set (operator over-loading, function overloading, that's about it), or I will just use straight up C. I like the simplicity of C and I find that when I use C++ I spend far too much brainpower on thinking about the language and not enough time thinking of the actual problem I'm trying to solve.

2) I'm not an expert on ECSes.

I wrote this because I heard about the idea of an ECS and thought they were cool but often explained poorly or overly complicated, so I wanted to explain my understanding in a way that made sense to me. I've not done anything else with ECSes since.

I hope you still find this useful but take it for what it is.

- **What is an ECS?**
- **The Entity**
- **The Component**
- **The Signature**
- **The Entity Manager**
- **The Component Array**
- **The Component Manager**
- **The System**
- **The System Manager**
- **The Coordinator**
- **Demo**
- **Conclusion**
- **Source Code**
- **Further Reading**

What is an ECS?

Traditionally in game development, you would follow an inheritance approach to problems. A **Goblin** inherits from a **Monster** which inherits from an **Actor**. A **Shopkeeper** inherits from a **Human** which also inherits from an **Actor**. The **Actor** class contains a function called **Render()** which knows how to render an **Actor**, so for every **Goblin** you can call **Goblin.Render()** and for every **Shopkeeper** you can call **Shoperkeeper.Render()**.

There are two main problems with this approach. The first is the problem of flexibility. If you decide that you want to visit a town of friendly goblins in the game, and you have **Goblin Shopkeepers**, your inheritance tree gets messed up. You have all of the shopkeeping functionality in the **Shopkeeper** class (selling, bartering, whatever), but your **Goblin Shopkeeper** can't inherit from **Shopkeeper** because that would make the **Goblin Shopkeeper** a **Human**. Without a

doubt, inheritance has its place in software development, but in gameplay programming it can cause problems.

The second problem is a misuse of the cache. In games, you commonly iterate over a set of objects multiple times per second, running methods on them every frame. For example, your physics system might iterate over all objects that are subject to physics and call **Object.Integrate(dt)**, updating their position, velocity, and acceleration. So traditionally you'd have your big object that contains all of its state, including those needed for physics, and you'd call the integrate function on every object that needs to be updated. In each object's **Integrate()** method, you access the object's position, velocity, and acceleration member variables. When you access position, it's pulled into a cache line along with nearby member variables. Some of those nearby member variables will be useful (velocity and the acceleration), while others will not be. This is a huge waste of the cache and in an age where the performance bottleneck is the time it takes for data to get from main memory to the CPU's memory, it's a big deal.

The tides have been shifting into component-based design to solve the first problem. Looking at Unity, for example, all of the game objects are component-based. You start with a blank object that has only the default required Transform component, and you add more components to give the object functionality. But that hasn't solved the second problem.

The second problem is solved by keeping all of the data that will be iterated upon regularly packed tightly into memory so that an entire cache line's worth of data can be loaded at once, and when the next item is iterated upon, its data is already in the cache. This is solved by defining components as Plain Old Data (POD), essentially a simple struct with only the relevant data included. To continue the physics example, you might have **Transform** with position, **Rigidody** with velocity and acceleration, and **Gravity** with the gravitational constant g.

The physics system would then iterate over all "objects" that "contain" these three components, pulling in only the data it cares about into the cache.

> Unity is moving in this direction with the introduction of its own ECS implementation, as well as its Jobs system and the Burst compiler. In fact, watching a talk by Mike Acton (Principal Programmer at Unity leading ECS develop-

ment) is what got me interested in this stuff in the first place.

In reality, the traditional concept of the "object" is gone. Instead we have an **Entity** which is simply an ID. It doesn't "contain" anything. Instead the ID is used as an index into an array of components. An array is contiguous in memory which lends itself well to being the data structure of choice. So the physics system might have a list of all entities that have a **Transform**, **RigidBody**, and **Gravity** component, and use the entity's ID as an index into the **Transform** array, into the **RigidBody** array, and into the **Gravity** array.

So conceptually it's all pretty simple. An **Entity** is an ID. A **Component** is a struct of data. A **System** is the logic that operates on the components. The meat of this post will be on how to implement those three elements in a way that is simple, easy to understand, and easy to use.

I set out to design mine with the following goals:

- Relatively simple and understandable
- Modern C++
- Minimal amount of if conditions (to avoid branch mispredictions)
- Minimal virtual inheritance (to avoid virtual function lookup and branch mispredictions)

The Entity

As promised, an **Entity** is very simple:

```
1    // A simple type alias
2    using Entity = std::uint32_t;
3
4    // Used to define the size of arrays later on
5    const Entity MAX_ENTITIES = 5000;
```

You could of course choose for an **Entity** to be of any size, and same with **MAX_ENTITIES**.

The Component

A component is almost as simple as an entity. It's just a struct with a small chunk of functionally related data. As an example, **Transform** might look like this:

```
struct Transform
{
    Vec3 position;
    Quat rotation;
    Vec3 scale;
}
```

Each component type (**Transform**, **RigidBody**, etc) also has a unique ID given to it (for reasons explained later).

```
// A simple type alias
using ComponentType = std::uint8_t;

// Used to define the size of arrays later on
const ComponentType MAX_COMPONENTS = 32;
```

Again, you could choose any size for **ComponentType** and **MAX_COMPONENTS**.

The Signature

Since an entity is simply an ID, we need a way to track which components an entity "has", and we also need a way to track which components a system cares about.

I chose the very simple approach of using a **std::bitset** (modern C++ equivalent of a bitfield), called a **Signature**. Each component type has a unique ID (starting from 0), which is used to represent a bit in the signature.

As an example, if **Transform** has type 0, **RigidBody** has type 1, and **Gravity** has type 2, an entity that "has" those three components would have a signature of 0b111 (bits 0, 1, and 2 are set).

A system would also register its interest in certain components as another signature. Then it's a simple bitwise comparison to ensure that an entity's signature contains the system's signature (an entity might have *more* components

than a system requires, which is fine, as long as it has *all* of the components a system requires).

```
1     // A simple type alias
2     using Signature = std::bitset<MAX_COMPONENTS>;
```

The Entity Manager

The **Entity Manager** is in charge of distributing entity IDs and keeping record of which IDs are in use and which are not.

I chose to use a simple **std::queue**, where on startup the queue is initialized to contain every valid entity ID up to **MAX_ENTITIES**. When an entity is created it takes an ID from the front of the queue, and when an entity is destroyed it puts the destroyed ID at the back of the queue.

```
class EntityManager
{
public:
    EntityManager()
    {
        // Initialize the queue with all possible e
        for (Entity entity = 0; entity < MAX_ENTITI
        {
            mAvailableEntities.push(entity);
        }
    }

    Entity CreateEntity()
    {
        assert(mLivingEntityCount < MAX_ENTITIES &&

        // Take an ID from the front of the queue
        Entity id = mAvailableEntities.front();
        mAvailableEntities.pop();
        ++mLivingEntityCount;

        return id;
    }

    void DestroyEntity(Entity entity)
    {
        assert(entity < MAX_ENTITIES && "Entity out

        // Invalidate the destroyed entity's signat
        mSignatures[entity].reset();

        // Put the destroyed ID at the back of the
        mAvailableEntities.push(entity);
        --mLivingEntityCount;
    }

    void SetSignature(Entity entity, Signature sign
    {
        assert(entity < MAX_ENTITIES && "Entity out

        // Put this entity's signature into the arr
        mSignatures[entity] = signature;
    }

    Signature GetSignature(Entity entity)
```

```
46          {
47              assert(entity < MAX_ENTITIES && "Entity out
48
49              // Get this entity's signature from the arr
50              return mSignatures[entity];
51          }
52
53      private:
54          // Queue of unused entity IDs
55          std::queue<Entity> mAvailableEntities{};
56
57          // Array of signatures where the index correspo
58          std::array<Signature, MAX_ENTITIES> mSignatures
59
60          // Total living entities - used to keep limits
61          uint32_t mLivingEntityCount{};
62      };
```

The Component Array

We need to create a data structure that is essentially a simple array, but is always a *packed* array, meaning it has no holes. If an entity is just an index into an array of components, then it's simple to grab the relevant component for an entity, but what happens when an entity is destroyed? That index into the array is no longer valid.

Remember that the entire point of the ECS is to keep the data packed in memory, meaning that you should be able to iterate over all of the indices in the array without needing any sort of "if(valid)" checks. When an entity is destroyed, the component data it "had" still exists in the arrays. If a system were to then try to iterate over the array, it would encounter stale data with no entity attached. For this reason we need to keep the array packed with valid data at all times.

I chose to solve this problem by keeping a mapping from entity IDs to array indices. When accessing the array, you use the entity ID to look up the actual array index. Then, when an entity is destroyed, you take the last valid element in the array and move it into the deleted entity's spot and update the map so that the entity ID now points to the correct position. There is also a map from the array index to an entity ID so that, when moving the last array element, you know which entity was using that index and can update its map.

Before showing the code, let me demonstrate the process visually because try as I might to make the code understandable, it's still clearer in picture form.

Let's assume that **MAX_ENTITIES** is set to 5. The array starts out empty, there is nothing in the maps, and the size is 0.

```
Start
------
Array: []

Entity->Index: []

Index->Entity: []

Size: 0
```

We then add a component with value A to Entity 0.

Entity 0 maps to Index 0, and Index 0 maps to Entity 0.

```
Add A to Entity 0
------
Array: [A]

Entity->Index:
[0:0] Entity 0's data (A) is at Index 0

Index->Entity:
[0:0] Index 0 holds Entity 0's data (A)

Size: 1
```

We then add a component with value B to Entity 1.

Entity 1 maps to Index 1, and Index 1 maps to Entity 1.

```

```

```
Add B to Entity 1
------
Array: [A, B]

Entity->Index:
[0:0] Entity 0's data (A) is at Index 0
[1:1] Entity 1's data (B) is at Index 1

Index->Entity:
[0:0] Index 0 holds Entity 0's data (A)
[1:1] Index 1 holds Entity 1's data (B)

Size: 2
```

We then add a component with value C to Entity 2.

Entity 2 maps to Index 2, and Index 2 maps to Entity 2.

```
Add C to Entity 2
------
Array: [A, B, C]

Entity->Index:
[0:0] Entity 0's data (A) is at Index 0
[1:1] Entity 1's data (B) is at Index 1
[2:2] Entity 2's data (C) is at Index 2

Index->Entity:
[0:0] Index 0 holds Entity 0's data (A)
[1:1] Index 1 holds Entity 1's data (B)
[2:2] Index 2 holds Entity 2's data (C)

Size: 3
```

We then add a component with value D to Entity 3.

Entity 3 maps to Index 3, and Index 3 maps to Entity 3.

```
```

```
Add D to Entity 3
------
Array: [A, B, C, D]

Entity->Index:
[0:0] Entity 0's data (A) is at Index 0
[1:1] Entity 1's data (B) is at Index 1
[2:2] Entity 2's data (C) is at Index 2
[3:3] Entity 3's data (D) is at Index 3

Index->Entity:
[0:0] Index 0 holds Entity 0's data (A)
[1:1] Index 1 holds Entity 1's data (B)
[2:2] Index 2 holds Entity 2's data (C)
[3:3] Index 3 holds Entity 3's data (D)

Size: 4
```

So far so good. Everything is packed into memory. But then we delete the value B from Entity 1. To keep it packed, we move the last element D into the spot occupied by B, and update the maps.

Entity 3 maps to Index 1, and Index 1 maps to Entity 3.

```
Delete B (which was at Index 1 and was the data of
------
Array: [A, D, C]

Entity->Index:
[0:0] Entity 0's data (A) is at Index 0
[3:1] Entity 3's data (D) is at Index 1
[2:2] Entity 2's data (C) is at Index 2

Index->Entity:
[0:0] Index 0 holds Entity 0's data (A)
[1:3] Index 1 holds Entity 3's data (D)
[2:2] Index 2 holds Entity 2's data (C)

Size: 3
```

We then delete the value D from Entity 3, moving the last element C into the spot occupied by D.

Entity 2 maps to Index 1, and Index 1 maps to Entity 2.

```
    Delete D (which was at Index 1 and was the data of
    ------
    Array: [A, C]

    Entity->Index:
    [0:0] Entity 0's data (A) is at Index 0
    [2:1] Entity 2's data (C) is at Index 1

    Index->Entity:
    [0:0] Index 0 holds Entity 0's data (A)
    [1:2] Index 1 holds Entity 2's data (C)

    Size: 2
```

Finally we add value E to Entity 4.

Entity 4 maps to Index 2, and Index 2 maps to Entity 4.

```
    Add E to Entity 4
    ------
    Array: [A, C, E]

    Entity->Index:
    [0:0] Entity 0's data (A) is at Index 0
    [2:1] Entity 2's data (C) is at Index 1
    [4:2] Entity 4's data (E) is at index 2

    Index->Entity:
    [0:0] Index 0 holds Entity 0's data (A)
    [1:2] Index 1 holds Entity 2's data (C)
    [2:4] Index 2 holds Entity 4's data (E)

    Size: 3
```

Voila, components removed and added while maintaining density.

```
```

```cpp
// The one instance of virtual inheritance in the e
// An interface is needed so that the ComponentMana
// can tell a generic ComponentArray that an entity
// and that it needs to update its array mappings.
class IComponentArray
{
public:
    virtual ~IComponentArray() = default;
    virtual void EntityDestroyed(Entity entity) = 0
};


template<typename T>
class ComponentArray : public IComponentArray
{
public:
    void InsertData(Entity entity, T component)
    {
        assert(mEntityToIndexMap.find(entity) == mE

        // Put new entry at end and update the maps
        size_t newIndex = mSize;
        mEntityToIndexMap[entity] = newIndex;
        mIndexToEntityMap[newIndex] = entity;
        mComponentArray[newIndex] = component;
        ++mSize;
    }

    void RemoveData(Entity entity)
    {
        assert(mEntityToIndexMap.find(entity) != mE

        // Copy element at end into deleted element
        size_t indexOfRemovedEntity = mEntityToInde
        size_t indexOfLastElement = mSize - 1;
        mComponentArray[indexOfRemovedEntity] = mCo

        // Update map to point to moved spot
        Entity entityOfLastElement = mIndexToEntity
        mEntityToIndexMap[entityOfLastElement] = in
        mIndexToEntityMap[indexOfRemovedEntity] = e

        mEntityToIndexMap.erase(entity);
        mIndexToEntityMap.erase(indexOfLastElement)

        --mSize;
```

```
46              --mSize;
47          }
48
49          T& GetData(Entity entity)
50          {
51              assert(mEntityToIndexMap.find(entity) != mE
52
53              // Return a reference to the entity's compo
54              return mComponentArray[mEntityToIndexMap[en
55          }
56
57          void EntityDestroyed(Entity entity) override
58          {
59              if (mEntityToIndexMap.find(entity) != mEnti
60              {
61                  // Remove the entity's component if it
62                  RemoveData(entity);
63              }
64          }
65
66      private:
67          // The packed array of components (of generic t
68          // set to a specified maximum amount, matching
69          // of entities allowed to exist simultaneously,
70          // has a unique spot.
71          std::array<T, MAX_ENTITIES> mComponentArray;
72
73          // Map from an entity ID to an array index.
74          std::unordered_map<Entity, size_t> mEntityToInd
75
76          // Map from an array index to an entity ID.
77          std::unordered_map<size_t, Entity> mIndexToEnti
78
79          // Total size of valid entries in the array.
80          size_t mSize;
81      };
```

The **unordered_map** does have a performance penalty because when you want to get the ID of a component to grab it from the contiguous array, you have to request it from the unordered_map which is not contiguous. An alternative would be to use arrays instead.

But the **unordered_map**s have the nice property of supporting **find()**, **insert()**, and **delete()**, which allow for asserting validity without "if(valid)" checks and it's a

bit clearer then setting array elements to some "INVALID" value.

The virtual inheritance of **IComponentArray** is unfortunate but, as far as I can tell, unavoidable. As seen later, we'll have a list of every **ComponentArray** (one per component type), and we need to notify all of them when an entity is destroyed so that it can remove the entity's data if it exists. The only way to keep a list of multiple templated types is to keep a list of their common interface so that we can call **EntityDestroyed()** on all of them.

Another method is to use events, so that every **ComponentArray** can subscribe to an Entity Destroyed event and then respond accordingly. This was my original approach but I decided to keep **ComponentArray**s relatively stupid.

Yet another method would be to use some fancy template magic and reflection, but I wanted to keep it as simple as possibe for my own sanity. The cost of calling the virtual function **EntityDestroyed()** should be minimal because it isn't something that happens every single frame.

The Component Manager

Now we can implement the **Component Manager**, which is in charge of talking to all of the different **ComponentArrays** when a component needs to be added or removed.

As mentioned earlier, we need to have a unique ID for every type of component so that it can have a bit in a signature. To accomplish that without pain, I have the Component Manager have a **ComponentType** variable that increments by one with every component type that is registered. I've seen implementations that don't require any sort of **RegisterComponent** functionality, but I've found it to be the simplest method. The downside is that any time you add a new type of component to your game and want to use it, you will first need to call **RegisterComponent**.

C++ offers a convenient function that will return a pointer to a const char array representation of a type T. That pointer (which is just an integer) can be used as a unique key into a map of **ComponentTypes**.

That same key is also used as a unique key into a map of **IComponentArray** pointers, so there is one **ComponentArray** instantiation per **ComponentType**.

```cpp
class ComponentManager
{
public:
    template<typename T>
    void RegisterComponent()
    {
        const char* typeName = typeid(T).name();

        assert(mComponentTypes.find(typeName) == mC

        // Add this component type to the component
        mComponentTypes.insert({typeName, mNextComp

        // Create a ComponentArray pointer and add
        mComponentArrays.insert({typeName, std::mak

        // Increment the value so that the next com
        ++mNextComponentType;
    }

    template<typename T>
    ComponentType GetComponentType()
    {
        const char* typeName = typeid(T).name();

        assert(mComponentTypes.find(typeName) != mC

        // Return this component's type - used for
        return mComponentTypes[typeName];
    }

    template<typename T>
    void AddComponent(Entity entity, T component)
    {
        // Add a component to the array for an enti
        GetComponentArray<T>()->InsertData(entity,
    }

    template<typename T>
    void RemoveComponent(Entity entity)
    {
        // Remove a component from the array for an
        GetComponentArray<T>()->RemoveData(entity);
    }

    template<typename T>
```

```
46          template<typename T>
47          T& GetComponent(Entity entity)
48          {
49              // Get a reference to a component from the
50              return GetComponentArray<T>()->GetData(enti
51          }
52
53          void EntityDestroyed(Entity entity)
54          {
55              // Notify each component array that an enti
56              // If it has a component for that entity, i
57              for (auto const& pair : mComponentArrays)
58              {
59                  auto const& component = pair.second;
60
61                  component->EntityDestroyed(entity);
62              }
63          }
64
65      private:
66          // Map from type string pointer to a component
67          std::unordered_map<const char*, ComponentType>
68
69          // Map from type string pointer to a component
70          std::unordered_map<const char*, std::shared_ptr
71
72          // The component type to be assigned to the nex
73          ComponentType mNextComponentType{};
74
75          // Convenience function to get the statically c
76          template<typename T>
77          std::shared_ptr<ComponentArray<T>> GetComponent
78          {
79              const char* typeName = typeid(T).name();
80
81              assert(mComponentTypes.find(typeName) != mC
82
83              return std::static_pointer_cast<ComponentAr
84          }
85      };
```

The System

A system is any functionality that iterates upon a list of entities with a certain signature of components.

Every system needs a list of entities, and we want some logic outside of the system (in the form of a manager to maintain that list), so I use a **System** base class that has only a **std::set** of entities.

I chose a **std::set** rather than a **std::list** for a few reasons.

First, each entity is unique, and a set is defined as having every element be unique, so it maps well logically.

Second, each entity is an integer which makes for easy compares when inserting/removing from the set. Removing a specific entity from a list is O(n) because you have to start at the beginning and possibly go to the end, while removing from a set is O(log n) because it's a binary tree. However, inserting into a list is only O(1) while inserting into a set is O(log n) as well.

Third, it makes the code easier to understand and read. With a list, you have to use **std::find** to check if an entity is in the list, but with **std::set** you can just call **insert()** and **erase()** directly without any checks. If trying to insert when it's already in, it does nothing. If trying to erase when it doesn't exist, it does nothing.

Fourth, I tested using a list and using a set, and a set was faster.

```
1    class System
2    {
3    public:
4        std::set<Entity> mEntities;
5    };
```

Each system can then inherit from this class which allows the System Manager (see next section) to keep a list of pointers to systems. Inheritance, but not virtual.

A system could then do something like this:

```
    for (auto const& entity : mEntities)
    {
        auto& rigidBody = GetComponent<RigidBody>(entit
        auto& transform = GetComponent<Transform>(entit
        auto const& gravity = GetComponent<Gravity>(ent

        transform.position += rigidBody.velocity * dt;

        rigidBody.velocity += gravity.force * dt;
    }
```

A **RigidBody**, a **Transform**, and a **Gravity** will be pulled into the cache for this entity as well as for all of the entities near it in the component array, which are likely to be needed with the next entity in the list of entities.

The System Manager

The **System Manager** is in charge of maintaining a record of registered systems and their signatures. When a system is registered, it's added to a map with the same **typeid(T).name()** trick used for the components. That same key is used for a map of system pointers as well.

As with components, this approach requires a call to **RegisterSystem()** for every additional system type added to the game.

Each system needs to have a signature set for it so that the manager can add appropriate entities to each systems's list of entities. When an entity's signature has changed (due to components being added or removed), then the system's list of entities that it's tracking needs to be updated.

If an entity that the system is tracking is destroyed, then it also needs to update its list.

```cpp
class SystemManager
{
public:
    template<typename T>
    std::shared_ptr<T> RegisterSystem()
    {
        const char* typeName = typeid(T).name();

        assert(mSystems.find(typeName) == mSystems.

        // Create a pointer to the system and retur
        auto system = std::make_shared<T>();
        mSystems.insert({typeName, system});
        return system;
    }

    template<typename T>
    void SetSignature(Signature signature)
    {
        const char* typeName = typeid(T).name();

        assert(mSystems.find(typeName) != mSystems.

        // Set the signature for this system
        mSignatures.insert({typeName, signature});
    }

    void EntityDestroyed(Entity entity)
    {
        // Erase a destroyed entity from all system
        // mEntities is a set so no check needed
        for (auto const& pair : mSystems)
        {
            auto const& system = pair.second;

            system->mEntities.erase(entity);
        }
    }

    void EntitySignatureChanged(Entity entity, Sign
    {
        // Notify each system that an entity's sign
        for (auto const& pair : mSystems)
        {
            auto const& type = pair.first;
```

```
46                  auto const& system = pair.second;
47                  auto const& systemSignature = mSignatur
48
49                  // Entity signature matches system sign
50                  if ((entitySignature & systemSignature)
51                  {
52                      system->mEntities.insert(entity);
53                  }
54                  // Entity signature does not match syst
55                  else
56                  {
57                      system->mEntities.erase(entity);
58                  }
59              }
60          }
61
62      private:
63          // Map from system type string pointer to a sig
64          std::unordered_map<const char*, Signature> mSig
65
66          // Map from system type string pointer to a sys
67          std::unordered_map<const char*, std::shared_ptr
68      };
```

The Coordinator

We now have quite a lot of functionality built up. We have entities which are managed by an **Entity Manager**. We have components which are managed by a **Component Manager**. And we have systems which are managed by a **System Manager**. These three managers also need to talk to each other.

There are a few ways of accomplishing that, such as having them all be globals, or using an event system, but I opted to instead bundle them into a single class called **Coordinator** (alternative name suggestions welcome) that acts as a mediator. This allows us to have a single instance of the coordinator (as a global or whatever you want), and we can use it to interface with all of the managers. It also makes usage easier because you can replace this:

```
1     Entity player = entityManager.CreateEntity();
2     componentManager.AddComponent<Transform>(player);
3     RenderSystem renderSystem = systemManager.RegisterSy
```

With this:

```
    Entity player = coordinator.CreateEntity();
    coordinator.AddComponent<Transform>(player);
    RenderSystem renderSystem = coordinator.RegisterSyst
```

The coordinator has pointers to each manager and does some meta-managing between them.

```
class Coordinator
{
public:
    void Init()
    {
        // Create pointers to each manager
        mComponentManager = std::make_unique<Compon
        mEntityManager = std::make_unique<EntityMan
        mSystemManager = std::make_unique<SystemMan
    }


    // Entity methods
    Entity CreateEntity()
    {
        return mEntityManager->CreateEntity();
    }

    void DestroyEntity(Entity entity)
    {
        mEntityManager->DestroyEntity(entity);

        mComponentManager->EntityDestroyed(entity);

        mSystemManager->EntityDestroyed(entity);
    }


    // Component methods
    template<typename T>
    void RegisterComponent()
    {
        mComponentManager->RegisterComponent<T>();
    }

    template<typename T>
    void AddComponent(Entity entity, T component)
    {
        mComponentManager->AddComponent<T>(entity,

        auto signature = mEntityManager->GetSignatu
        signature.set(mComponentManager->GetCompone
        mEntityManager->SetSignature(entity, signat

        mSystemManager->EntitySignatureChanged(enti
```

```
46          }
47
48          template<typename T>
49          void RemoveComponent(Entity entity)
50          {
51              mComponentManager->RemoveComponent<T>(entit
52
53              auto signature = mEntityManager->GetSignatu
54              signature.set(mComponentManager->GetCompone
55              mEntityManager->SetSignature(entity, signat
56
57              mSystemManager->EntitySignatureChanged(enti
58          }
59
60          template<typename T>
61          T& GetComponent(Entity entity)
62          {
63              return mComponentManager->GetComponent<T>(e
64          }
65
66          template<typename T>
67          ComponentType GetComponentType()
68          {
69              return mComponentManager->GetComponentType<
70          }
71
72
73          // System methods
74          template<typename T>
75          std::shared_ptr<T> RegisterSystem()
76          {
77              return mSystemManager->RegisterSystem<T>();
78          }
79
80          template<typename T>
81          void SetSystemSignature(Signature signature)
82          {
83              mSystemManager->SetSignature<T>(signature);
84          }
85
86      private:
87          std::unique_ptr<ComponentManager> mComponentMan
88          std::unique_ptr<EntityManager> mEntityManager;
89          std::unique_ptr<SystemManager> mSystemManager;
90      };
```

that acts as a wrapper to an ID with methods that call into the **EntityManager** and **ComponentManager** directly (e.g., **entity.RemoveComponent()**), which makes for more intuitive usage, but I found it to cause the code to be more complicated and hard to understand. I tried to do it that way multiple times but each time came upon recursive header issues. In the end I opted for the cleaner but less intuitive **Coordinator**.

Demo

Now let's see how all of this might be used in a demo that instantiates 10,000 cubes and then has them fall under the influence of gravity. We'll ignore the rendering and the math classes because that isn't what this post is about, but keep in mind there is also a rendering system and a Vec3 class.

We have the following components:

```
struct Gravity
{
    Vec3 force;
};

struct RigidBody
{
    Vec3 velocity;
    Vec3 acceleration;
};

struct Transform
{
    Vec3 position;
    Vec3 rotation;
    Vec3 scale;
};
```

A system for rudimentary physics integration:

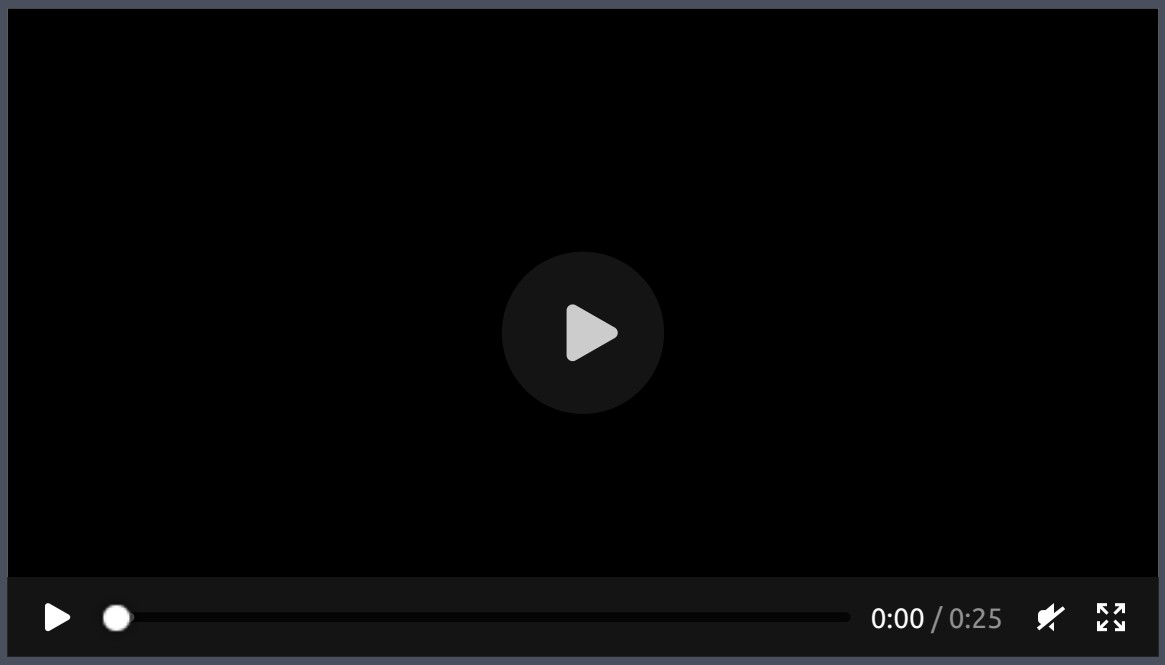
```
1     extern Coordinator gCoordinator;
2
3     void PhysicsSystem::Update(float dt)
4     {
5         for (auto const& entity : mEntities)
6         {
7             auto& rigidBody = gCoordinator.GetComponent
8             auto& transform = gCoordinator.GetComponent
9             auto const& gravity = gCoordinator.GetCompo
10
11            transform.position += rigidBody.velocity *
12
13            rigidBody.velocity += gravity.force * dt;
14        }
15    }
```

Then the main loop:

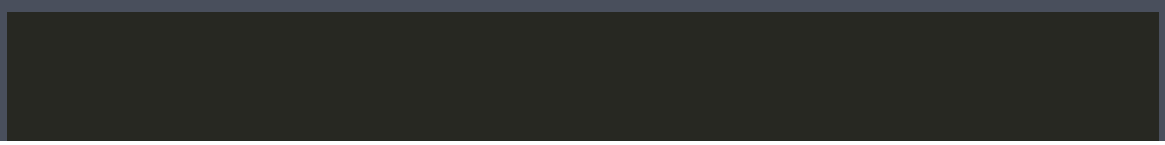
```cpp
Coordinator gCoordinator;

int main()
{
    gCoordinator.Init();

    gCoordinator.RegisterComponent<Gravity>();
    gCoordinator.RegisterComponent<RigidBody>();
    gCoordinator.RegisterComponent<Transform>();

    auto physicsSystem = gCoordinator.RegisterSyste

    Signature signature;
    signature.set(gCoordinator.GetComponentType<Gra
    signature.set(gCoordinator.GetComponentType<Rig
    signature.set(gCoordinator.GetComponentType<Tra
    gCoordinator.SetSystemSignature<PhysicsSystem>(

    std::vector<Entity> entities(MAX_ENTITIES);

    std::default_random_engine generator;
    std::uniform_real_distribution<float> randPosit
    std::uniform_real_distribution<float> randRotat
    std::uniform_real_distribution<float> randScale
    std::uniform_real_distribution<float> randGravi

    float scale = randScale(generator);

    for (auto& entity : entities)
    {
        entity = gCoordinator.CreateEntity();

        gCoordinator.AddComponent(
            entity,
            Gravity{Vec3(0.0f, randGravity(generato

        gCoordinator.AddComponent(
            entity,
            RigidBody{
                .velocity = Vec3(0.0f, 0.0f, 0.0f),
                .acceleration = Vec3(0.0f, 0.0f, 0.
            });

        gCoordinator.AddComponent(
            entity,
```

```
46                  Transform{
47                      .position = Vec3(randPosition(gener
48                      .rotation = Vec3(randRotation(gener
49                      .scale = Vec3(scale, scale, scale)
50                  });
51          }
52
53          float dt = 0.0f;
54
55          while (!quit)
56          {
57              auto startTime = std::chrono::high_resoluti
58
59              physicsSystem->Update(dt);
60
61              auto stopTime = std::chrono::high_resolutio
62
63              dt = std::chrono::duration<float, std::chro
64          }
65      }
```

Here is the result:



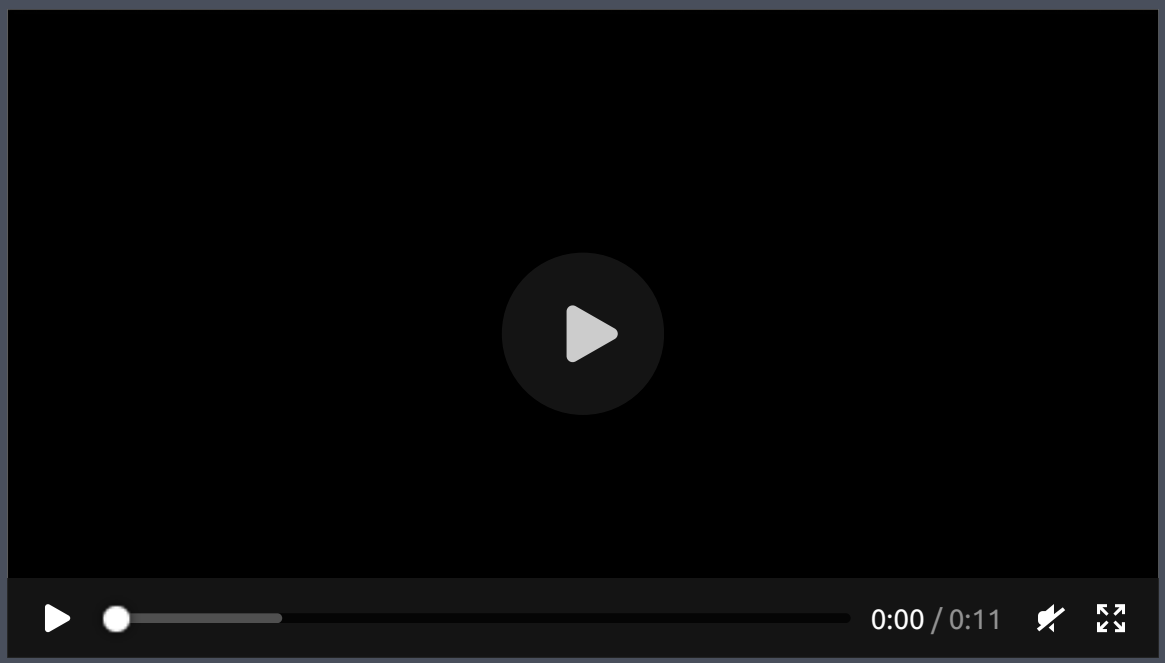
Here is the output from Valgrind's cachegrind tool if you're curious:



```
==15445== I   refs:      3,632,270,619
==15445== I1  misses:       87,147,982
==15445== LLi misses:           26,599
==15445== I1  miss rate:          2.40%
==15445== LLi miss rate:          0.00%
==15445==
==15445== D   refs:      1,583,125,924  (1,045,689,
==15445== D1  misses:       11,968,989  (    7,776,
==15445== LLd misses:          505,598  (      270,
==15445== D1  miss rate:           0.8% (
==15445== LLd miss rate:           0.0% (
==15445==
==15445== LL refs:          99,116,971  (   94,924,
==15445== LL misses:           532,197  (      297,
==15445== LL miss rate:            0.0% (
```

This is a very simple example of course, but it's still fun.

Circling back to the beginning about components also making complex behavior easier, we could easily flip our demo by not adding a **RigidBody** or a **Gravity** to the cubes, and instead adding them to the camera.



That is the camera falling down while the cubes remain still.

Conclusion

If you were skeptical about the idea of an ECS, I hope I've convinced you it has its merits. And if you were confused about how to implement one (like I was for a long time), I hope I've helped you find a way.

Source Code

You can find all of the source code **here**.

> All of the ECS-related source code exists only in headers for two reasons. First, there are a lot of templates which must be in headers anyway. Second, it may possibly increase the odds that the compiler will inline.

Further Reading

- **1**
- **2**
- **3**
- **4**

Last Edited: Dec 20, 2022